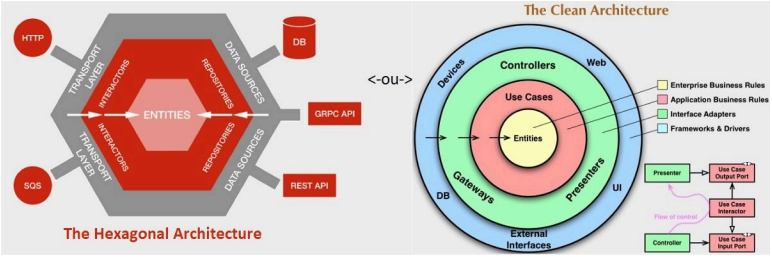


Hexagonal or Clean Architecture

Wellington Guimarães Pimenta
Full Stack Development | Tech Leadership | Cloud Architecture | Financial systems | Microservices | Agile methodologies | .N...

10 de novembro de 2023

Artigo baseado no artigo da DEV Community https://dev.to/dyarleniber/hexagonal-architecture-and-clean-architecture-with-examples-48oi

Hexagonal Architecture and Clean Architecture

Uma das coisas que mudou minha carreira como desenvolvedor de software e mudou minha perspectiva sobre como construir software foi o conhecimento de como arquitetar minhas aplicações e como projetar meu código de uma forma mais profissional, permitindo que minhas aplicações sejam escalonadas.

Continue lendo para entender melhor os conceitos e porque acredito que a melhor maneira de entender a Arquitetura Limpa é entender primeiro a Arquitetura Hexagonal

Normalmente quando começamos a desenvolver software, principalmente quando utilizamos algum framework, tendemos a usar sempre aquele conhecido padrão MVC, dividindo nossa aplicação em modelos, visualizações e controladores.

Isso não é tão ruim se você está construindo algo simples, como um MVP por exemplo, porém como precisamos escalar esta aplicação, adicionar mais funcionalidades, substituir uma biblioteca, ou algo parecido, enfrentamos vários problemas pois conforme nosso código tende para ficar mais acoplado fica cada vez mais difícil fazer alterações em nossa aplicação sem precisar alterar também muitos lugares diferentes em nosso código.

Com isso nosso software fica mais suscetível a bugs e desmotiva todos os desenvolvedores que precisam manter aquele código. E isso é extremamente comum, principalmente se você está começando, normalmente não somos ensinados a arquitetar nossa aplicação, geralmente deixamos algum framework tomar a decisão por nós, e acabamos colocando toda a lógica de negócio dentro do controlador, por exemplo.

Lembro-me de fazer muito isso quando estava começando minha carreira e de ter sido rejeitado em muitas entrevistas, principalmente para empresas que procuravam alguém de nível mais sênior ou médio.

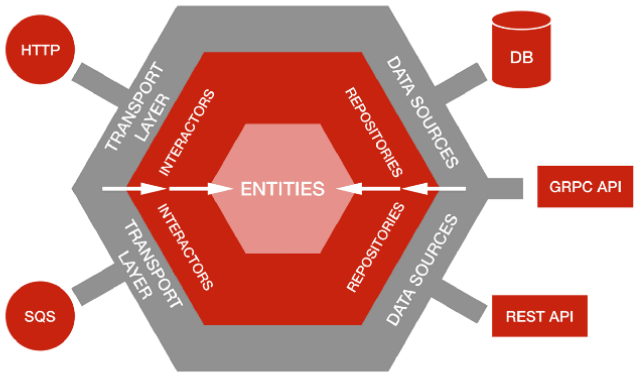
Dito isso, meu principal objetivo neste artigo é dar a você uma melhor compreensão de como arquitetar sua aplicação, separando o software em camadas, e como organizar e isolar sua lógica de negócios, para que você não fique mais espremido tudo dentro do controlador.

Arquitetura Hexagonal

(Arquitetura de Portas e Adaptadores)

Antes de mergulharmos na Arquitetura Limpa, primeiro daremos uma breve olhada na Arquitetura Hexagonal (ou Arquitetura de Portas e Adaptadores, que acredito ser um nome melhor).

A Arquitetura Hexagonal (também conhecida como Arquitetura de Portas e Adaptadores) é uma arquitetura de software baseada na ideia de isolamento da lógica de negócios central de preocupações externas, separando o aplicativo em componentes fracamente acoplados.



Vamos imaginar que estamos construindo uma API que precisa buscar alguns dados de uma API REST.

Em vez de ter nossa classe de caso de uso (ou classe de serviço) fortemente acoplada à API REST externa, podemos simplesmente criar uma Porta, que define a interface que nossa classe de caso de uso precisa implementar para buscar os dados (independentemente de como os dados são buscado, usando uma API REST, um banco de dados, uma API GraphQL, etc.). E então podemos criar a Adapter, que é uma classe que implementa a interface Port e delega as chamadas para a API REST externa (ou outro serviço externo).

Dessa forma, se precisarmos alterar a forma como buscamos os dados, bastamos criar um novo Adapter que implemente a interface Port, sem precisar alterar a implementação do caso de uso.

"A ideia da Arquitetura Hexagonal é colocar entradas e saídas nas bordas do nosso design. A lógica de negócios não deve depender de expormos uma API REST ou GraphQL, e não deve depender de onde obtemos os dados – um banco de dados, uma API de micros serviço exposta via gRPC ou REST, ou apenas um simples arquivo CSV."

"O padrão nos permite isolar a lógica central da nossa aplicação de preocupações externas. Ter nossa lógica central isolada significa que podemos alterar facilmente os detalhes da fonte de dados sem um impacto significativo ou grandes reescritas de código na base de código."

É importante ressaltar que a Arquitetura Hexagonal veio antes da Arquitetura Limpa, porém ambas compartilham o mesmo objetivo, que é a separação de interesses. Na verdade, a Arquitetura Hexagonal foi um dos padrões arquitetônicos que Robert C. Martin (Uncle Bob - autor de Arquitetura Limpa) utilizou como referência a Arquitetura Limpa.

Porém, a Arquitetura Hexagonal carece de alguns detalhes de implementação, é aí que entra a Arquitetura Limpa, para preencher essas "lacunas".

Exemplo de typescript

Vamos dar uma olhada na arquitetura hexagonal em ação usando TypeScript.

Uma das maneiras de criar essas portas e adaptadores é usando o Princípio de Inversão de Dependência, o último princípio SOLID (e isso é uma espécie de spoiler para Arquitetura Limpa).

O **Princípio de Inversão de Dependência** afirma que:

"Módulos de alto nível não devem importar nada de módulos de baixo nível. Ambos devem depender de abstrações (por exemplo, interfaces)."

"As abstrações não devem depender de detalhes. Os detalhes (implementações concretas) devem depender de abstrações."

Suponhamos que queremos criar uma classe de casos de uso para exportar um usuário.

Se você não está familiarizado com o conceito de casos de uso, não se preocupe, abordaremos isso com mais detalhes na seção Arquitetura Limpa.

Mas, por enquanto, lembre-se de que uma classe de caso de uso (ou classe de serviço na terminologia Domain-Driven Design) é uma classe responsável por uma ação (intenção) específica do usuário. Como buscar um pedido, criar um produto, atualizar um usuário, excluir um artigo, etc.

E normalmente, uma classe controladora (responsável por lidar com as solicitações do usuário) invoca a classe de caso de uso para executar a ação do usuário.

E a maneira mais comum de fazer isso é ter a classe de caso de uso (ou mesmo o controlador) fortemente acoplada à lógica de exportação.

Mas ao invés de fazer isso, podemos criar uma interface para abstrair a lógica de exportação (Porta):

```
type User = {
    name: string,
    email: string,
    dateOfBirth: Date
};

// Port interface
interface ExportUser {
    export(user: User);
}
```

E então crie uma (ou mais) implementações desta interface (Adaptador):

```
// CSV Adapter implementation
class ExportUserToCSV implements ExportUser {
    export(user) {
        // Export user to a CSV file
    }
}

// PDF Adapter implementation
class ExportUserToPDF implements ExportUser {
    export(user) {
        // Export user to a PDF file
    }
}
```

Fazendo isso, a implementação da exportação do usuário dependerá da interface (abstração), conforme afirma o Princípio da Inversão de Dependência.

E a classe de caso de uso não dependerá de nenhuma implementação concreta:

```
// Use case class
class ExportUserUseCase {
    constructor(private exportUser: ExportUser) {}

    execute(user: User) {
        this.exportUser.export(user);
    }
}
```

Finalmente, podemos alternar entre diferentes implementações da exportação do usuário, sem precisar alterar a implementação da classe de caso de uso:

```
// Export user to a CSV file
const exportUserToCSV = new ExportUserToCSV();
const exportUserUseCase = new ExportUserUseCase(exportUserToCSV);

exportUserUseCase.execute({
    name: 'John Doe',
    email: 'john.doe@mail.com',
    dateOfBirth: new Date()
});
```

```
// Export user to a PDF file
const exportUserToPDF = new ExportUserToPDF();
const exportUserUseCase = new ExportUserUseCase(exportUserToPDF);

exportUserUseCase.execute({
    name: 'John Doe',
    email: 'john.doe@mail.com',
    dateOfBirth: new Date()
});
```

Arquitetura Limpa (Clean Architecture)

Agora, depois de dar uma olhada na Arquitetura Hexagonal, podemos começar a ver melhor do que se trata a Arquitetura Limpa.

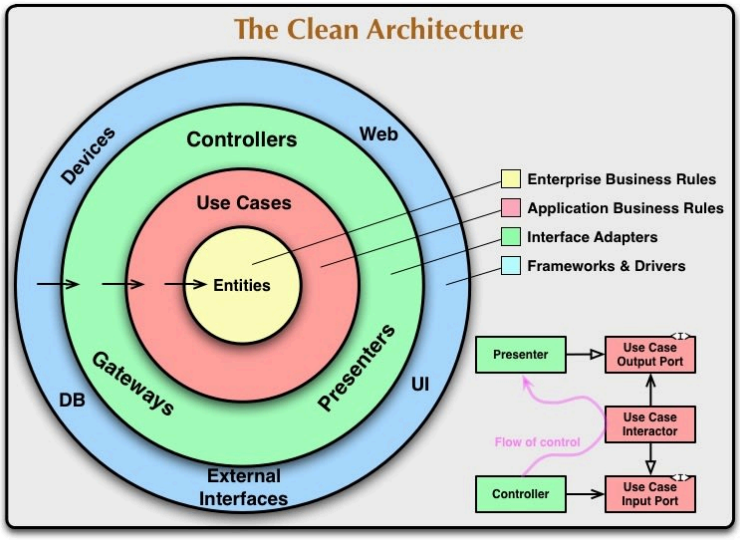
A Arquitetura Hexagonal e muitos outros padrões arquitetônicos compartilham o mesmo objetivo: a separação de interesses. E todos atingem esse objetivo dividindo a aplicação em camadas. E a **Clean Architecture é apenas uma tentativa de integrar todas essas arquiteturas em uma única ideia**.

É importante mencionar que a Arquitetura Limpa NÃO é apenas uma estrutura de pastas que você pode copiar e colar no seu projeto.

É a ideia de separar a aplicação em camadas e obedecer à Regra da Dependência, criando um sistema que seja:

- Independente de Estruturas
- Testável
- Independente da IU
- Independente de banco de dados
- Independente de qualquer agência externa

Para começar, vamos percorrer as camadas da Arquitetura Limpa:



Entidades (camada de domínio)

Esta camada é responsável pela lógica de negócios da aplicação. É a camada mais estável e basicamente o coração do aplicativo.

É aqui que podemos aplicar algumas táticas de Design Orientado a Domínio, como Agregados, Objetos de Valor, Entidades, Serviços de Domínio, etc. A propósito, o título do livro que introduziu o conceito de Domain-Driven Design em 2004 é **"Design Orientado a Domínio: Enfrentando a Complexidade no Coração do Software"**, de Eric Evans.

Esta camada é isolada do resto do aplicativo (preocupações com as camadas externas).

Casos de uso (camada de aplicação)

Esta camada contém as regras de negócios específicas do aplicativo. Implementa todos os casos de uso da aplicação, utiliza as classes de

domínio, mas está isolado dos detalhes e implementação de camadas externas, como bancos de dados, adaptadores, etc.

Esta camada contém apenas interfaces para interagir com o mundo exterior.

Como vimos antes, os casos de uso são as ações ou intenções do usuário, como buscar um pedido, criar um produto e assim por diante. E cada caso de uso deve ser independente dos outros casos de uso, para estar em conformidade com o Princípio de Responsabilidade Única (o primeiro princípio SOLID).

Exemplo:

```
import { GetPostByIdUseCase } from '@application/interfaces/use-cases/posts/GetPostByIdUseCase';
import { GetPostByIdRepository } from '@application/interfaces/repositories/posts/GetPostByIdRepository';
import { PostNotFoundError } from '@application/errors/PostNotFoundError';

export class GetPostById implements GetPostByIdUseCase {
    constructor(
        private readonly getPostByIdRepository: GetPostByIdRepository,
    ) {}

    async execute(postId: GetPostByIdUseCase.Input): Promise<GetPostByIdUseCase.Output> {
        const post = await this.getPostByIdRepository.getPostById(postId);
        if (!post) {
            return new PostNotFoundError();
        }
        return post;
    }
}
```

Neste exemplo, temos uma interface (GetPostByIdUseCase) que contém Input e Output do caso de uso, que são os DTOs (Data Transfer Objects), que são usados em conjunto com outras interfaces (como o GetPostByIdRepositório interface) para interagir com o mundo exterior.

Observe que este caso de uso é responsável apenas por buscar uma postagem pelo seu ID, caso precise deletar uma postagem, precisamos criar outro caso de uso. É muito comum trabalhar com o padrão MVC, por exemplo, ter uma classe de controlador gigantesca que trata de todas as solicitações do usuário. No entanto, fere o Princípio da Responsabilidade Única.

Adaptadores de interface (camada de infraestrutura)

A camada Infraestrutura é a camada que contém todas as implementações concretas da aplicação, como os repositórios, os adaptadores, as conexões de banco de dados, etc.

Estruturas e drivers

Esta camada é composta por frameworks e ferramentas como o Banco de Dados, o Web Framework, etc. Normalmente, não escrevemos muito código nesta camada.

Apenas Quatro camadas?

Não existe nenhuma regra que diga que você deve sempre ter apenas essas quatro camadas. No entanto, a regra da dependência sempre se aplica.

A regra da dependência

A Regra de Dependência afirma que:

"As dependências do código-fonte só podem apontar para dentro."

"Nada em um círculo interno pode saber alguma coisa sobre algo em um círculo externo."

Basicamente, o nome de algo declarado em um círculo externo NÃO deve ser mencionado pelo código em um círculo interno. Isso inclui funções, classes. variáveis ou qualquer outra entidade de software nomeada.

Não queremos que nada no círculo externo tenha impacto nos círculos internos.

Por exemplo, na camada de domínio NÃO devemos mencionar nada dentro da camada de aplicação, ou dentro da camada de infraestrutura. O mesmo vale para a camada de aplicação, NÃO devemos mencionar nada dentro da camada de infraestrutura e assim por diante.

Mas obviamente, podemos usar as entidades definidas na camada de domínio na camada de aplicação, por exemplo.

Considerações finais.

Clean Architecture é um ótimo padrão de arquitetura a ser seguido se você deseja projetar um aplicativo em larga escala (e ir além das simples CRUDs com o padrão MVC).

Isso o ajudará a escrever um código limpo, flexível, testável e de fácil manutenção, separando o aplicativo em camadas.

Além disso, se você deseja melhorar ainda mais a organização de sua lógica de negócios e a modelagem do domínio de sua aplicação, também pode aproveitar as vantagens do Domain-Driven Design. Arquitetura Limpa quando combinada com Design Orientado a Domínio é uma ferramenta realmente poderosa!

Limpa.

abaixo o github com o codigo completo deste projeto:

GitHub repository of this project.

Divirta-se !

Wellington Guimarães Pimenta

Nenhum comentário ainda.

Seja a primeira pessoa a comentar.

Dê início à conversa

Gostou deste artigo?

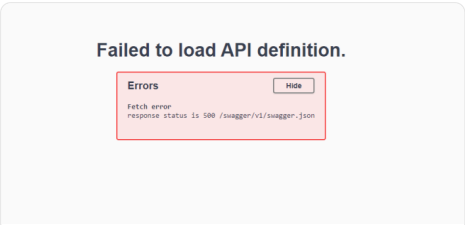
Siga para nunca perder uma atualização.

Wellington Guimarães Pimenta

Full Stack Development | Tech Leadership | Cloud Architecture | Financial systems | Microservices | Agile methodologies | .Net Specialist | SQL Senior Developer

Seguir

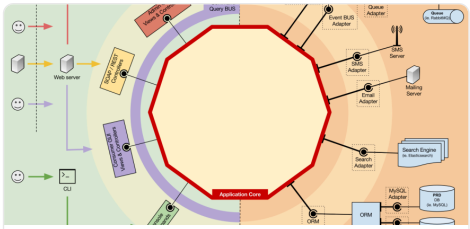
Mais artigos para você



O Desafio de Mapear Controllers com Swagger em Projetos .NET sem um Padrão Arquitetural

Diego Baumbach

3



Clean Architecture - Parte 1

Michel Torres

42 · 2 compartilhamentos

Published on LinkedIn

Artigo Clean Architecture

Rafael Bonamigo

1